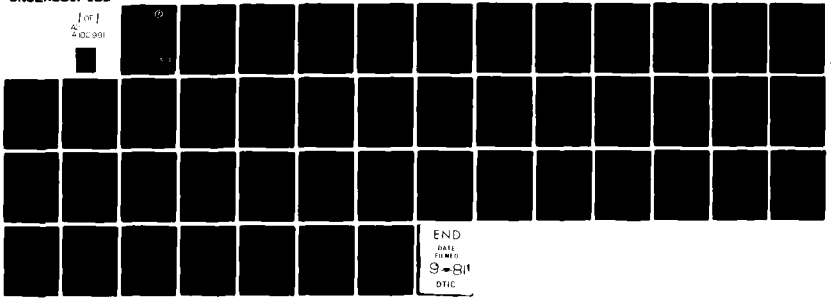
AD-A102 991 INSTITUTE FOR SOFTWARE ENGINEERING PALO ALTO CA F/G 9/2
FOUNDATIONS AND CONCEPTS OF SOFTWARE PHYSICS,(U)
APR 79 R P KOVACH, K W KOLENCE N00014-78-C-0768
UNCLASSIFIED NL



FOUNDATIONS AND CONCEPTS OF SOFTWARE PHYSICS

by

R. P. Kovach and K. W. Kolence

April, 1979

Institute for Software Engineering
P.O. Box 637, Palo Alto, CA 94303

ABSTRACT

FOUNDATIONS AND CONCEPTS OF SOFTWARE PHYSICS

R. P. Kovach and K. W. Kolence

Kolence's theory of the performance of computing systems is restated. In this development, the basic subsystems are reduced to two: logical subconfigurations and software units. These and the three fundamental variables, work, time and storage occupancy are all defined using as a basis a single logical construct, the set of instantaneous descriptions of instruction executions. Basic results are given for time relationships (utilizations and concurrency levels), work relationships (distribution numbers), and work and time relationships (absolute power and relative power) are derived. Interpretation of variables used in other approaches in software physics terms are indicated.

This work was supported in part by the Office of Naval Research under Contract number N00014-78-C-0768

INTRODUCTION

Software physics is a theory for characterizing the behavior of executing computing systems named and proponed chiefly by K. W. Kolence. Its fundamental concepts, basic results and a variety of applications have been given in a number of publications by him, (KOLE70, KOLE72, KOLE73, KOLE75a, KOLE75b, KOLE76). The fullest and most recent treatment of the theory and some of its applications is given in KOLE76. The main motivation of the book, and the theory itself, arises from the many problems inherent in capacity management functions in data processing organizations. At this time a considerable body of work exists, based on software physics, in such areas as workload forecasting and capacity planning, equipment and configuration planning, performance management, cost accounting and charging policy and budgeting. So far this work has withstood well the many tests provided by the rough and tumble world of present day computer installations.

The present paper is nothing more than a restatement of the nuclear theory as presented by Kolence. It is done in order to better meet the accepted requirements of theory construction: economy of assumptions and basic (undefined) terms, more compact and rigorous derivation of results and more logical and orderly development based on established principles and methods.

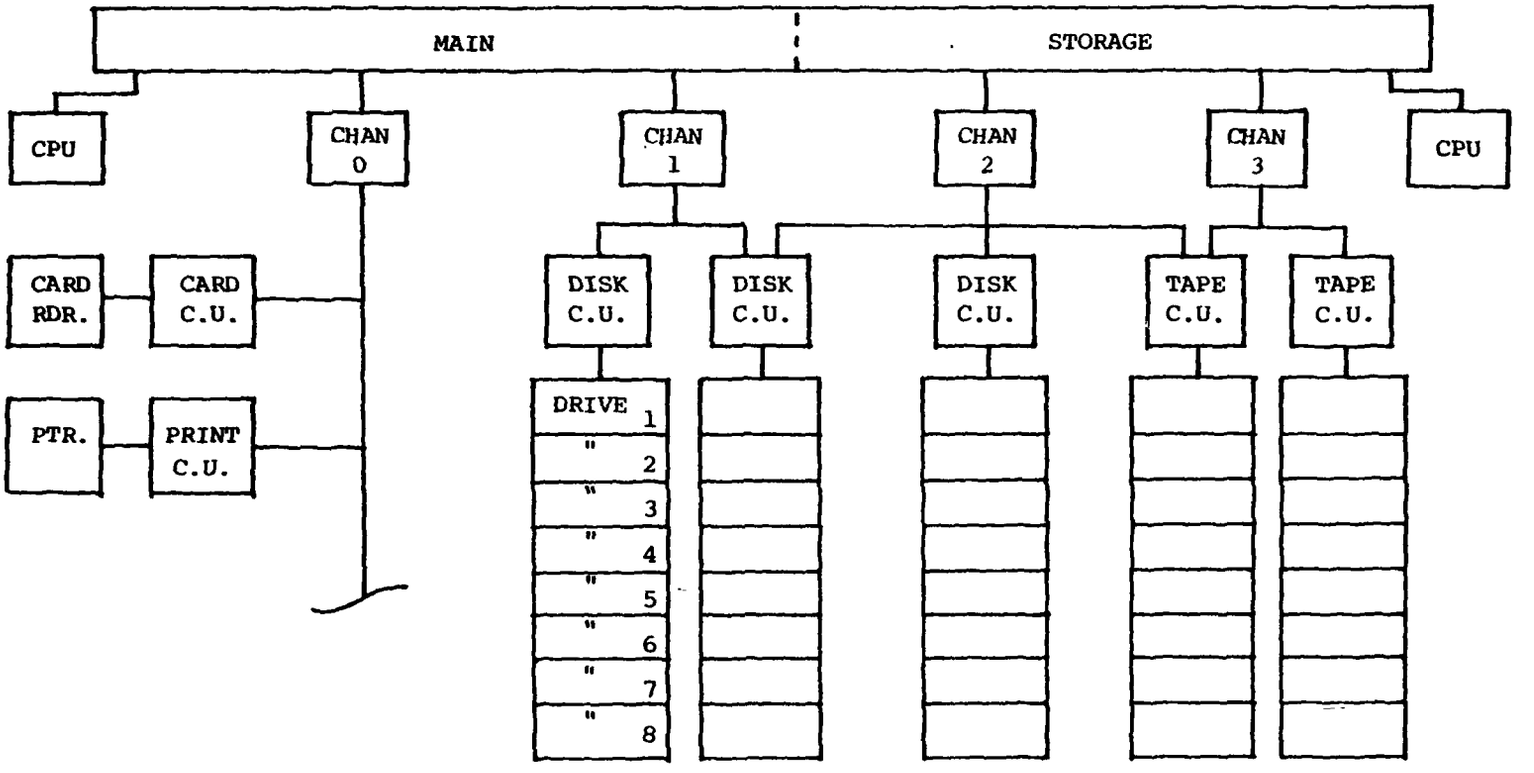
In the present development only one (assumed) logical construct is required, namely, a complete set of complete instantaneous descriptions of instruction executions. From this, using set theoretic (including graph theoretic) methods, the two basic subsystems, logical subconfigurations and software units, are defined. Then the three fundamental variables, work, time and storage occupancy are defined. Relationships among these are determined by the structural properties of software unit/logical subconfiguration pairs. Kolence's notation has been modified slightly and extended. A few results, not previously educed, are given.

The only other attempts at a coherent approach to this subject are queueing theory and what's now known as the operational analysis of queueing networks. In this paper there are a few anticipatory hints on interpretations of software physics variables and relationships leading to queueing theoretic results. This whole area is fully explored and developed by Traister (TRAI79). From the point of view of the queueing analyst the basic theory of software physics analyzes service times into their constraints. In a practical setting this is important, revealing what variables may be changed and what the consequences are for service times. The use of software work in place of service requests introduces a weighting factor to requests more accurately reflecting the demand placed on the system by requests. The introduction of system related clocks and the relationships among their timings adds a new dimension to the characterization of utilizations, service times and response times. The indications at this time are that not only can software physics subsume queueing network analysis as applied to computer systems but that a whole new set of results will emerge from the combined approach.

Figure 1 shows a conventional representation of a computer system configuration. There is a wide variety of charting conventions employed in drawing diagrams such as this but they all have the same essential properties, the differences arising from considerations of ease of maintenance of the drawing, symbol conventions, local traditions and so forth. Such graphs are very useful for showing all the devices in an installation, the cabling, logical and physical addresses and similar hardware related data.

When analyzing the dynamic behavior of computing systems, however, these diagrams may be deceptive because they ignore the influence of time and time sequence on the topology of the system. False assumptions may be made unconsciously, leading to erroneous conclusions about the relationship between configuration and performance characteristics. In the example at hand, for instance, it appears that there are two paths between main storage and the disk drives in the second string, one via channel 1 and one via channel 2. At the time of any given execution, however, a drive may be connected to main storage by only one channel, either channel 1 or channel 2. With time taken into consideration, all such apparent alternate paths are mutually exclusive.

It is convenient to think of the usual configuration diagram as the graph union of all possible paths that may occur in the course of some execution. With that in mind then, appearances notwithstanding, these configuration graphs are rooted trees. There is one path between any higher node and any lower one and there are relative roots at every level. Most important for present consideration is that they exhibit the upper lattice property, that is, every node contains or covers the properties of each node dependent from it. So, for instance, we say that channel 1 is a disk channel because it contains control units which contain disk drives. For the same reason, channel 2 is a disk channel. Channel 3 is a tape channel because it contains

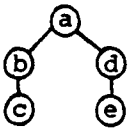


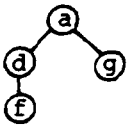
A CONFIGURATION - "STANDARD" REPRESENTATION

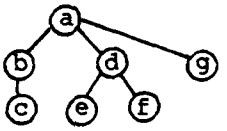
FIGURE 1

control units which contain tape drives. Channel 2 is a tape channel for the same reason.

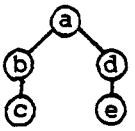
At this point we require a definition for a logical subconfiguration. This in turn requires a definition of a composition operation on graphs which we call a graft. To form a graft of several trees, take their graph union. If the resultant graph has a root then the graph is complete. For example, the union of:

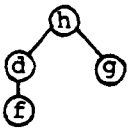
A = 

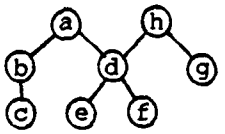
and B = 

is A ∪ B = 

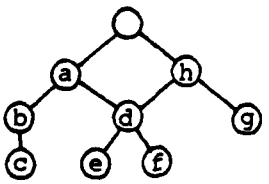
which is a graft. If the resultant graph does not have a unique root, then one is created and all the relative roots of the union are made immediate descendents of it. For example, if the union of:

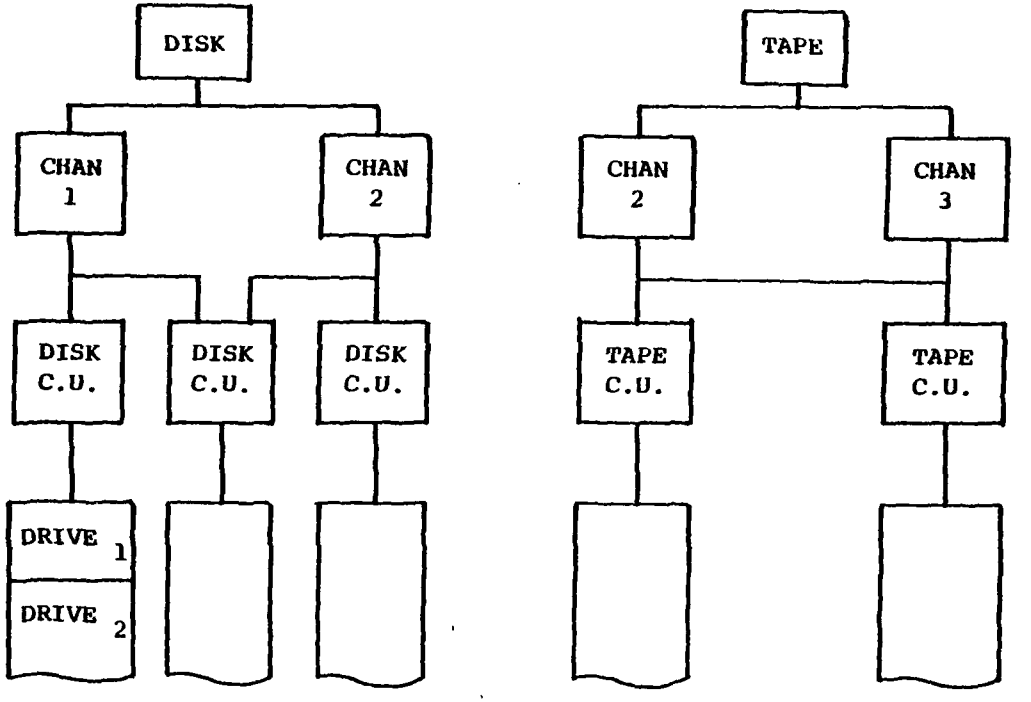
A = 

and C = 

is A ∪ C = 

which produces the graft:

A ∧ C = 

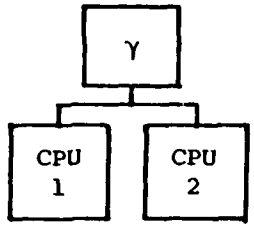


THE DISK SUBCONFIGURATION

THE TAPE SUBCONFIGURATION

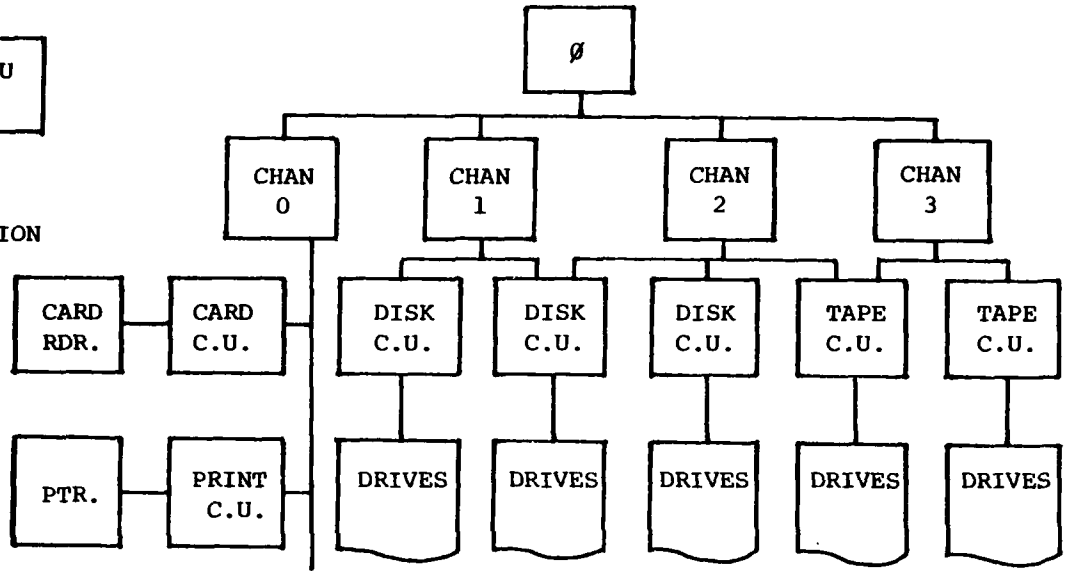
EQUIPMENT CLASS SUBCONFIGURATIONS

FIGURE 2



THE CPU

SUBCONFIGURATION

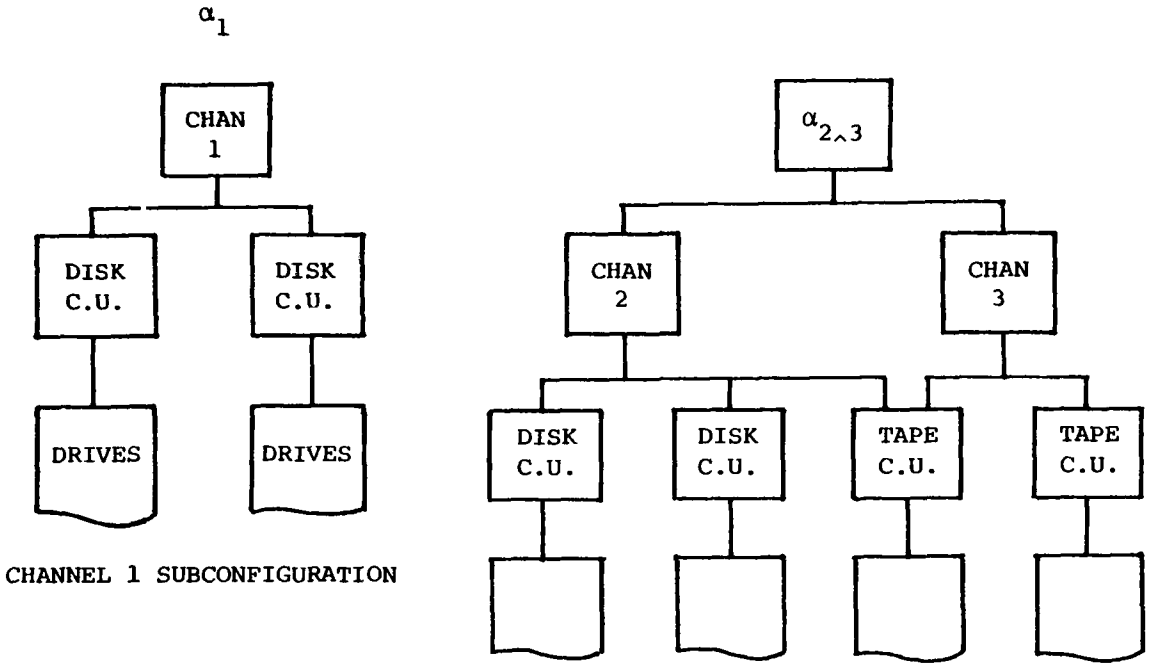


THE INPUT/OUTPUT SUBCONFIGURATION

(EQUIPMENT CLASS = PERIPHERAL DRIVES)

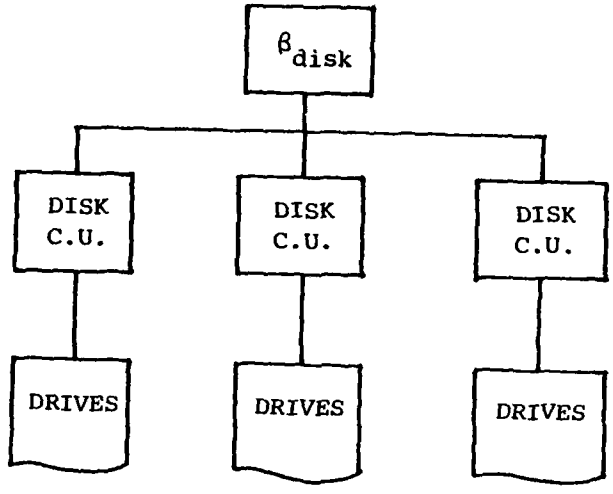
EQUIPMENT CLASS SUBCONFIGURATIONS

FIGURE 3



CONFIGURATION LOGICAL SUBCONFIGURATIONS

FIGURE 4



DISK CONTROL UNIT SUBCONFIGURATION

COMBINED EQUIPMENT CLASS AND CONFIGURATIONAL LOGICAL SUBCONFIGURATION

FIGURE 5

A logical subconfiguration is a graft of configuration subtrees selected by either some set of properties that they contain or by the root nodes or, occasionally, by a combination of both. The most commonly used logical subconfigurations of the first kind are called equipment class subconfigurations. Some of them are illustrated in figures 2 and 3 based on the configuration of figure 1. Some examples of the second kind of logical subconfiguration, based on configurational characteristics are shown in figure 4. Note also that the logical subconfiguration made up of all channel subconfigurations would produce $\emptyset$, the Input/Output subconfiguration. Figure 5 shows a combined case where the root nodes are specified (control unit) and the equipment class selector (disk) as well.

Typically, the only subconfigurations with more than one level are the various I/O subconfigurations. These multiple levelled structures are supplied numerical identifiers, in the form of subscripts, by the conventional method used for numbering nodes of trees, one number for the first level, two numbers for the second level and so on. Thus drive number 4 on control unit number 2 on channel number 3 is δ_{324}. The control unit subconfiguration is designated β_{32} and the channel subconfiguration, α_3. This is, of course, the exact analog of the physical and logical addressing scheme used both in hardware the software systems. Note also that in the conventional configuration diagram certain devices (drives especially) may have more than one designation. This causes no inconsistency whatever because at any given time an action occurs on a device under only one designation. In other words, the actions going on under the several designations for the device are mutually exclusive with respect to time. Furthermore, all of our instrumentation will so report it. The fact that a single hardware device has several logical designations has no effect on the logical structure, it only places a constraint on the domain of possible simultaneous events.

The previous discussion and the figures mentioned there require an explanation of the Greek letters. Certain logical subconfigurations are so frequently used that it is convenient to have a compact notation to refer to them. The equipment classes are indicated with a brief word or abbreviation which is self-explanatory, e.g., disk,

tape, ptr, card, etc. Configurational logical subconfigurations are written as a single Greek letter, e.g., ψ is the whole configuration, $\emptyset$ is the I/O subconfiguration and γ is the CPU subconfiguration. For those subconfigurations where the root is actually a device, a Greek letter designates the subconfiguration and the corresponding Latin letter the device which is the root node. E.g., α_2 is the channel 2 subconfiguration and a_2 is the channel device which is channel 2, β_{21}, is the first control unit subconfiguration in the α_2 subconfiguration and b_{21}, is the control unit device which is its root node.

PROCESSORS AND STORAGES

In the conventional configuration diagram of figure 1, with the exception of the main storage, all of the devices depicted are processors. In general, all devices are either processors or storage. They are distinguished by their operational relationship in an execution: processors change the contents of storage. A rough rule for distinguishing the two is if the device may be programmed (microprogrammed) then it is a processor, if its contents are changed by a processor, it is a storage.

Clearly, processors contain storages and storages have built-in processors to control them. For instance, a channel may contain memories (registers) for such things as commands, addresses, error detection codes, status codes, etc. those memories may be manipulated by a microprocessor which itself contains scratch-pad memories and internal registers. This bifurcation can be recursively applied right down to the movement of electrons across molecular boundaries, if there is reason to do so. When we are examining activity at one level of processors and storages, the activity on storages by processors within them are considered internal, not part of the activity we are studying. Typically, internal activity is treated as a loss or degradation factor much like friction in mechanics. For example, if we are interested in a channel's activity in handling data transfers for a computation in the CPU, then the channel work involved in channel command processing, status checking and so on is considered internal work. If, on the other hand, we are interested in the activity on the status registers, address registers, etc. of the channel, then the activity of the microprocessor manipulating those registers is the external work and the activity within the microprocessor on its scratch pads and registers is internal work. These distinctions make the problem manageable, allowing us to isolate important variables at one level (ignoring the lower levels) and then reapply those concepts at lower levels much like the laws of mechanics discovered at the macroscopic level (ignoring friction) were applied to the molecular level to account for friction.

It is not necessary for a processor to be a physically distinguishable device, some free-standing box sitting out on the computer room floor. As long as a set of processor functions, exclusive of other processor functions can be distinguished, as long as a mental boundary can be put around a processor subsystem, then it may be considered a processor. So, for instance, a built-in channel, even when it shares circuitry with the CPU so that some of their execution times are mutually exclusive, is still a distinguishable processor. Furthermore, as we shall see, all of the relationships of fundamental variables that apply to processors individually also apply, with appropriate accommodation, to logical subconfigurations of processors, making them in effect logical processors.

All of the common equipment classes may be classified readily into processors or storage. A partial list is given below for reference and classification.

Processors:

- CPU
- Disk Drives
- Tape Drives
- Drum Drives
- Printers
- Card Readers
- Card Punches
- Paper Tape Reader
- Paper Tape Punches
- Terminals
- Mass Storage Drives
- MICR Readers
- A/D and D/A Converters
- Low Speed Channels
  (e.g., byte MPX)
- High Speed Channels
  (e.g., selector, Blk MPX)
- Disk Control Units
- Tape Control Units
- Drum Control Units

Storages:

- Main Storage (e.g., core, diode)
- Registers (everywhere)
- Disk (platters)
- Tape (reels)
- Drum
- Paper
- Cards
- Paper Tape
- Cartridges (mass store)
- Etc.

Printer Control Units
Card Control Units
Paper Tape Control Units
MICR Control Units
Transmission Control Units
Terminal Control Units
Mass Storage Control Units
Etc.

The reader may have noticed that one whole category of device has not been mentioned so far, namely cables, transmission lines, busses and the like. From the point of view of configurations they are treated only as connectors, arcs between nodes. They only enter into consideration when their capacity has some limiting effect on the performance characteristics of processors. Analysis of their capacity is most appropriately handled by considering them as channels in the information theoretic sense.

SOFTWARE UNITS

In describing the properties of an executing computing system or any of its subsystems it is necessary to specify not only the hardware constituent (device or subconfiguration) but also a "software" constituent. Clearly, this is so because a processor event consists of the execution of an instruction by a processor and the kind of properties we are interested in may be conditioned by the instruction and therefore, sets of events by sets of instructions.

The usual methods of designating or categorizing software are not adequate for our purposes. Most of them imply identities or equivalencies which simply will not hold in the situations we are describing. For instance, if the software constituent were defined to be a program in its source language state then a program compiled and executed on two different computers would be considered the same, in some regards at least. Similarly, a program run through two different compilers and executed on one machine or two different executions of the object code on one machine would be considered identical when, in fact, they might (rather, probably would) produce different series of processor events. Furthermore, such approaches limit scope, preventing us from talking about execution characteristics of sets of instructions across program boundaries.

To arrive at a more satisfactory definition we resort to the following heuristic device. Assume the existence (or the possibility of existence) of a complete collection of instantaneous descriptions of all instruction executions over some time interval, in the manner of an instruction execution trace. Each description contains a variety of information such as all relevant addresses of the instruction (absolute memory address, offsets from base registers, from page start, page number, etc.), the instruction itself, all operand addresses, the operands themselves (data), the time at the start and end of the execution and whatever else is required. More precisely, we should say assume that the information provided by such a trace is known. Now we are in a position to define a <u>software unit</u>.

A software unit is any subset taken from the complete collection of instruction-operand executions over some time interval.

Note that this means the processor events, the execution of instructions operating on operands and those operands, not some selected lines from a print-out.

The software unit (subset) consisting of the full set for some time interval is called the full workload for that interval and is denoted by L. If the time interval is partitioned into several intervals, then full workload for each subinterval is called a sub-workload, designated as L_i and $L = \bigcup_i L_i$.

Obviously, any software unit may be decomposed into other software units and the union or intersection of any two software units is also a software unit. Some decompositions or compositions are more usually employed than others. Most often software units are partitioned:

$$S = \bigcup_i S_i \text{ where } S_i \cap S_j = \emptyset \text{ for } i \neq j.$$

Some common partitionings are into software units where each unit is all the executions of a given instruction or class of instructions, into applications, jobs, and program runs (discussed below), into subworkloads as described above or into other partitions within sub-workloads and so on. Situations arise, especially in measurement experiments and in accounting procedures, where a decomposition is not known with certainty to be a partition. It is necessary to be aware of this fact so that false conclusions about properties of the aggregation may be avoided.

In a deterministic machine a software unit is completely determined by a set of instructions (program) and the full set of initial operand values (inputs) to them. Therefore, it is possible to treat the pair instruction-set/inputs as equivalent to a software unit. In practice, the only situations where both may be precisely specified is for applications, jobs, steps, program runs and the like. On occasion, following common parlance, expressions such as "sort work" and "work of the operating system" are used. The first of these means the work of the software unit made up of executions of the

sort program against some known set of inputs (last month's invoices, etc.) and similarly for the second. What is not meant, most emphatically, is that the sort program or the operating system program is a software unit.

NOTATION

In analyzing the behavior of executing computing systems we need to characterize sets of processor events and their consequences. A processor event always requires a "linked pair", an element of a software unit (executed instruction and operands) and a processor, including subconfigurations. To specify sets of events we must specify software units on sets of processors. Notationally, this is handled by using functional notation as it is commonly used in probability theory and combinatorial analysis. Software physics variables will generally be of the form:

F(software unit list; processor list)

where F is the functor, the two lists are separated by a semicolon and the elements of each list are separated by commas. Note that F is not a function of the software unit and subconfiguration. It is a variable or a value over the range specified in the two lists. The lists specify to what the variable or value pertains.

Certain classes of software units and certain logical subconfigurations are so frequently used that a standard set of symbols for them has been established for the sake of brevity and convenience. As described before, L is the software unit (subset) made up of all the instruction-operand executions for some period of observation. If L is partitioned by time, then the subworkloads are designated by L_i, i = 1, 2, 3 . . .. S is the symbol for any software unit whatsoever (including L) and S_i, i = 1, 2, 3 . . . is any subset of S.

Usually we will be interested in partitions of S, that is, where $S_i \cap S_j = \emptyset$ for $i \neq j$. If we need to distinguish hierarchies of subsets (partitions), the subscripting will be handled as it is for subconfigurations, described below.

The most common subconfigurations referred to are the whole configuration, ψ, the I/O subconfiguration, $\emptyset$, the CPU subconfiguration, γ, channel subconfigurations, α_i, control unit subconfigurations, β_{ij} and drives δ_{ijk}. For those subconfigurations which are hardware subconfigurations, that is, where the root node is an actual

device, it is desirable to have a distinct, but related, symbol for the device. The most usual are a_i for channels and b_{ij} for control units. The drives, being leaves of the tree, are degenerate cases of subconfigurations where the drives and the subconfigurations are one and the same thing, so no distinct symbol is required for the device.

In many cases it is necessary to indicate several software units or several subconfigurations (or both). Furthermore, it is always the case that an item in the list is contained by another item in the list (except the last, of course). Our convention will be to write the lower level or smaller item to the left. So now the format of a typical variable is:

F(software unit, containing S.U., . . .; subconfiguration, containing subconfiguration, . . .)

For example, we may be interested in the following utilization: $U(S_1,S;\gamma,\psi)$. This means the ratio of the time that software unit S_1 is executing on the CPU with respect to the time the containing software unit S is executing on the full configuration.

The symbol for any subconfiguration whatsoever is χ (chi). In writing many expressions it is necessary to not only indicate that one subconfiguration contains another but also how many levels away the other is. This is handled by the device of using an $_*$ in the subscripts. An asterisk means a string of one or more subscript values. The usual rules for substitution apply. Once a string of subscripts has been assigned to $_*$ then it must be substituted uniformly throughout the expression. Thus, if we write an expression about χ and χ_* then we are talking about any subconfiguration and any subconfiguration contained in it. If we write about χ and χ_{**} then we are talking about any subconfiguration at least two levels below χ. If we write $\sum_j F(S;\chi_{*j},\chi)$ then we are talking about the sum of the F's for all subconfigurations at some level which is at least two below χ, and so on. If it is required to deal with hierarchies of software units (especially partitions) then the same mechanism may be employed.

FUNDAMENTAL VARIABLES: TIME

Extent in time is the most generally perceived property of processor events. So much so, in fact, that it is commonly and erroneously used as a measure of the amount of activity of computing systems. On top of this there is considerable confusion over appropriate measures of time, especially for subconfigurations, and the relationship between device times and subconfiguration times. Clarity and precision in concept and definition of time is absolutely essential if the theory is to be fruitful and not break down at some later point in the deduction process.

The most basic measure of time is that of the time it takes a device to execute an instruction. This includes all of the time required by that device alone to perform the instruction, i.e., it is not just transfer time. It does not include preparation time, hold-off time or blocking time due to some other (perhaps internal) device even though the subject device is "connected" or signalling "busy." To illustrate, the execution time for a disk instruction includes the seek time, search time (rotational delay, latency) and the transfer time. It does not include the time after the seek is completed during which the search cannot start (if any), commonly called seek-delay, or the time consequent on RPS "misses" despite the fact the the drive is "busy" throughout all those times. Seek delay and RPS miss time are called delay times because they are due to some other drive blocking at the control unit or channel level. They are not due to the execution of the observed drive.

The execution time of device Y for software unit S is the total of the instruction execution times on Y of the instructions of S for Y. The functor for execution time is the symbol Tx and the above statement is written Tx(S;Y). On occasion, we shall need to discuss delay times and busy times. These are symbolized as Td and Tb respectively and the three are related by

$$Tb(S;Y) = Tx(S;Y) + Td(S;Y).$$

We extend the definition of execution time to subconfigurations by appealing to a concept common in modern physics: time changes in a system only when an event occurs in that system. Translated into software physics terms the definition states that execution time for a subconfiguration increases when any constituent of the subconfiguration is in execution. An equivalent, and perhaps more intuitively appealing definition may be made by using the notion of associated clocks. Each device has a clock and that clock runs whenever that device is executing. Associated with each node of every logical subconfiguration is a clock that runs when the clock of the device at that node (if there is one) is running or when the clock of any contained node is running. This approach has the same recursive character as the first one.

Because of the possibility of simultaneous activity at lower levels it is evident that the execution time of a subconfiguration is not, in general, the simple sum of the execution times of lower level subconfigurations. In fact,

$$Tx(S;\chi) \leq \sum_j Tx(S;\chi_j),$$

which also may be applied recursively. Likewise, dually,

$$Tx(S;\chi) \leq \sum_i Tx(S_i;\chi),$$

even when the S_i constitute a partition. To find a rule of composition for execution times, we must go back to the basics. Indicate a point in time when χ_j is executing by $tx(S;\chi_j)$. An instruction execution, somewhere in χ_j, generates a continuous point set, mappable to the real line, over some interval I. Indicate that point set by

$$\{tx(S;\chi_j)\}_I.$$

The amount of time to perform the instruction is a measure function on that point set (e.g., upper bound minus lower bound) indicated by

$$[\{tx(S;\chi_j)\}_I].$$

This in turn leads to

$$Tx(S;\chi_j) = \sum_I [\{tx(S;\chi_j)\}_I],$$

which is to say that the execution of χ_j for S is the sum of the time interval lengths for the execution of instructions from S for χ_j, as we said above.

Now, if we have several subconfigurations, χ_j (for various j), then the union of their point sets will create a new set of intervals, K, and we have

$$Tx(S;\chi) = \sum_K [\{\bigcup_j \{tx(S;\chi_j)\}\}_K].$$

This is the rule of composition we were seeking.

That forbidding looking nest of brackets is really asserting something that is in the realm of common sense. Some examples from figure 6 will make the point clear. Figure 6 illustrates an execution time pattern for three mutually exclusive software units, called S_1, S_2 and S_3, on a configuration consisting of one CPU, one channel with two control units, each with three disk drives. The whole time span has been divided into 100 units to permit times to be equated to ratios (percents) readily. The first band of three lines shows the execution time pattern of the three software units on the CPU. The next band of three lines shows the execution times of the channel and control unit devices. The next six lines show the time patterns for the drives, the labels above each segment showing the software unit associated with that execution. Several bands further down are the execution patterns for the two control unit subconfigurations and the channel subconfiguration, $Tx(L;\beta_{11})$, $Tx(L;\beta_{12})$ and $Tx(L;\alpha_1)$.

The execution pattern for δ_{110} is a broken set of intervals whose lengths total 55 units. Likewise, the sums of the interval lengths for δ_{111} and δ_{112} are 30 and 32.5 respectively. Now, form the "union" of those execution patterns (imagine the "ceiling" is lighted and look for the shadows on the "floor") and we get the execution pattern of β_{11}, which is just three execution intervals,

totalling 85 units. Do the same for the other three drives and β_{12}, producing one long execution interval of 80 units. Do the same with either all 6 drives or the two β execution patterns and produce the pattern for α_1. This, in a nutshell, was what the previous discussion was describing.

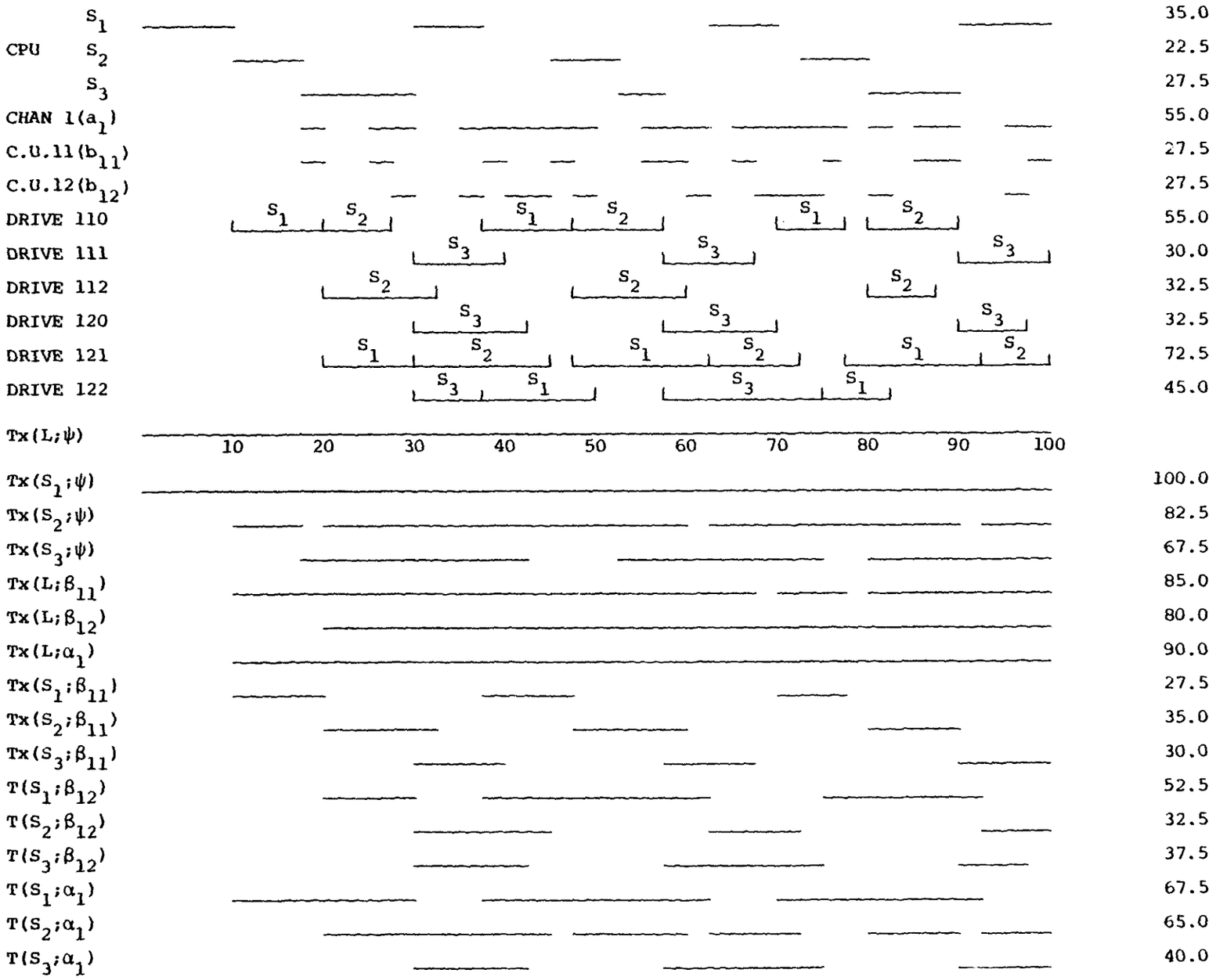

S_1 35.0
CPU S_2 22.5
S_3 27.5
CHAN 1(a_1) 55.0
C.U.11(b_{11}) 27.5
C.U.12(b_{12}) 27.5
DRIVE 110 55.0
DRIVE 111 30.0
DRIVE 112 32.5
DRIVE 120 32.5
DRIVE 121 72.5
DRIVE 122 45.0
Tx(L;ψ)
10 20 30 40 50 60 70 80 90 100
Tx(S_1;ψ) 100.0
Tx(S_2;ψ) 82.5
Tx(S_3;ψ) 67.5
Tx(L;β_{11}) 85.0
Tx(L;β_{12}) 80.0
Tx(L;α_1) 90.0
Tx(S_1;β_{11}) 27.5
Tx(S_2;β_{11}) 35.0
Tx(S_3;β_{11}) 30.0
T(S_1;β_{12}) 52.5
T(S_2;β_{12}) 32.5
T(S_3;β_{12}) 37.5
T(S_1;α_1) 67.5
T(S_2;α_1) 65.0
T(S_3;α_1) 40.0

FIGURE 6

One other observation should be made at this point. If the times shown for the drives were busy times rather than execution times, that is, included seek delay and RPS miss time as well as execution time, it would have no effect on the time pattern for the β or χ configurations. In other words,

$$Tb(L;\beta_{11}) = Tx(L;\beta_{11}) \text{ and } Tb(L;\alpha_1) = Tx(L;\alpha_1).$$

This is so because the delays experienced by one drive are due to the execution of some other drive. There is another class of time measure commonly encountered which should be accounted for in our system, elapsed time and the closely related idle time. It is clear from the definitions that idle time for a subconfiguration will not be registered on its own clock since that clock is only running when the subconfiguration is executing, i.e., not idle. Since elapsed time would be identical to execution time unless provision were made for intervening intervals of idle time, the same argument holds for elapsed time as well. Therefore, both must be measured with respect to some higher clock. Elapsed time for a subconfiguration χ_* is the difference, on the clock of χ, between the beginning of the earliest execution (least lower bound) and the end of the latest execution (greatest upper bound) of χ_*. Elapsed time for subconfigurations is written $Te(S;\chi_*,\chi)$. There is an analogous definition for the software unit dual, written $Te(S_i,S;\chi)$ and the composite case is $Te(S_i,S;\chi_*\chi)$.

One last definition, to specify the upper bound to the process:

$$Te(L;\psi,\psi) = Tx(L;\psi).$$

This assertion not only provides closure but it also states that there is no higher clock than that of $(L;\psi)$. In other words, idle times for the entire system are excluded from consideration. Periods when the whole system is idle because of lack of jobs, schedule problems, power outages, machine or software breakdowns or because the installation is closed for week-ends and holidays may be of interest or concern to management but they have nothing to do with a theory of executing computing systems.

The illustration of figure 6 should clarify some of the relationships discribed above. For example, $Te(L;\delta_{111},\beta_{11}) = 65$ because the beginning is at 30 and end at 100 but there are five units

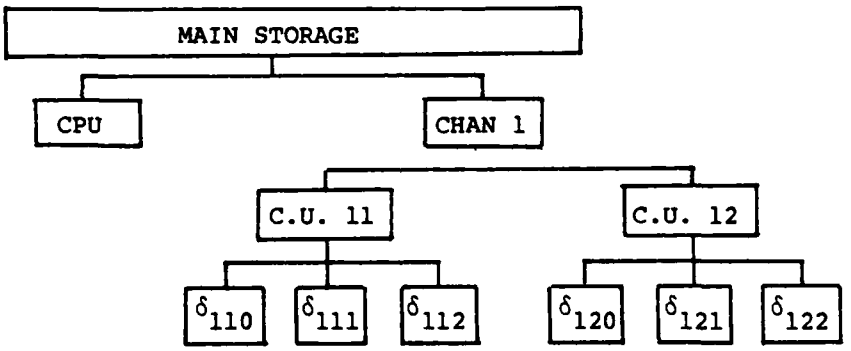
wherein the clock of β_{11} is not running (ending at 70 and 80). However, $Te(L;\delta_{111},\alpha_1) = 70$. Also, $Te(S_1,L;\delta_{121}) = 65$ but $Te(S_1,L;\beta_{11}) = 72.5$.

It is the usual, if not universal, practice to only use the highest clock, that of $(L;\psi)$ and to talk of elapsed time only with respect to software units. Why this should be, other than the force of habitual thinking, is not clear.

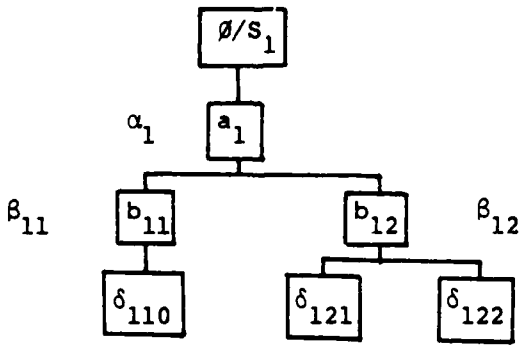
LOGICAL SUBCONFIGURATIONS AND SOFTWARE UNITS: DUALITY

In the previous discussion of logical subconfigurations no mention was made of the effects of software unit specifications on them, that subject not yet having been introduced. Since no software units were specified and the processor subconfiguration was considered to the union of all possible path graphs, in effect, we were assuming that the software unit was some full workload, L. The instantaneous path descriptions may be considered part of the information provided by the assumed trace. If we select some software unit, a subset of the executed instruction/operand pairs, then we are selecting a subset of the path descriptions as well.

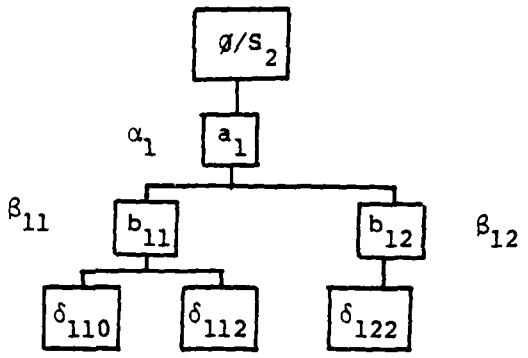
The conventional configuration underlying the time relationships illustrated in figure 6 is:



The logical subconfiguration $\emptyset$ for software unit S_1 is:



while $\emptyset$ for S_2 is:



and so on.

If S contains S_i then a subconfiguration χ/S contains χ/S_i just as subconfigurations based on devices do, e.g., if χ contains χ_* then χ/S contains χ_*/S. Therefore, any theorems about properties of executing systems arising from containment of device based subconfigurations will be true, suitably rewritten, for subconfigurations based on software units and subunits. This will be, in fact, most theorems.

It may have been noticed that subconfigurations based on software subunits have the same height as the containing subconfiguration. This will always be the case. Subconfigurations contained in device subconfigurations have a lower height. The first variety are formed by pruning leaves and branches, the second by removing roots. If these actions are considered as operations, then we can talk of the composition of the operations. Clearly, the order of performing the operations is immaterial, yielding the same resultant subconfiguration.

TIME RELATIONS: UTILIZATIONS AND CONCURRENCIES

Given that we have clocks at every node of every logical subconfiguration, it is natural to relate the readings of one clock to another. Two of the most common classes of relationship are the time on a clock with respect to that of a containing node and time on a clock with respect to that of a similar node in another subconfiguration. The first relationship yields utilization numbers and the second time-balance ratios, which can be derived from utilizations.

The definition of utilization for subconfigurations is:

$$U(S;\chi_*,\chi) = \frac{Tx(S;\chi_*)}{Tx(S;\chi)}.$$

This is the utilization of χ_* with respect to χ, the portion of the time that χ is executing when χ_* is also executing. On those occasions where we need to talk about utilizations based on busy time, Tb, or delay time, Td, then we will write Ub and Ud respectively. U is understood to mean Ux. Some examples from figure 6:

$$U(L;\delta_{110},\beta_{11}) = \frac{55}{85} = .65$$

$$U(S_2;\delta_{112},\beta_{11}) = \frac{32.5}{35} = .93$$

$$U(L;\delta_{121},\alpha_1) = \frac{72.5}{90} = .81 \text{ and so on.}$$

The dual for software units is:

$$U(S_i,S;\chi) = \frac{Tx(S_i;\chi)}{Tx(S;\chi)}.$$

this measure is commonplace in practice where χ is the CPU and S the full workload, i.e.,

$$U(S_i,L;\gamma) = \frac{Tx(S_i;\gamma)}{Tx(L;\gamma)}.$$

Examples from figure 6:

$$U(S_3,L;\beta_{11}) = \frac{30}{85} = .35$$

$$U(S_1,L;\psi) = \frac{100}{100} = 1.00, \text{ etc.}$$

The utilization resulting from the composition of operations is:

$$U(S_i,S;\chi_*,\chi) = \frac{Tx(S_i;\chi_*)}{Tx(S;\chi)},$$

that portion of time when S is executing on χ that S_i is executing on χ_*. Now, if we multiply the right side by:

$$\frac{Tx(S_i;\chi)}{Tx(S_i;\chi)}$$

and rearrange factors, we have:

$$U(S_i,S;\chi_*,\chi) = \frac{Tx(S_i;\chi_*)}{Tx(S_i;\chi)} \cdot \frac{Tx(S_i;\chi)}{Tx(S;\chi)}$$

$$= U(S_i;\chi_*,\chi) \cdot U(S_i,S;\chi)$$

Or, we can multiply by $\frac{Tx(S;\chi_*)}{Tx(S;\chi_*)}$ and have:

$$U(S_i,S;\chi_*,\chi) = \frac{Tx(S_i;\chi_*)}{Tx(S;\chi_*)} \cdot \frac{Tx(S;\chi_*)}{Tx(S;\chi)}$$

$$= U(S_i,S;\chi_*) \cdot U(S;\chi_*,\chi)$$

Rearranging the factors in one of these and comparing the two emphasizes the dual symmetry. Again, examples from figures 6:

$$U(S_3,L;\beta_{12},\alpha_1) = \frac{37.5}{90} = .42$$

$$U(S_3;\beta_{12},\alpha_1) = \frac{37.5}{40} = .94$$

$$U(S_3,L;\alpha_1) = \frac{40}{90} = .44$$

and $.42 \simeq .94 \times .44$, etc.

Because of the ratio definition there is an evident "chain rule":

$$U(S;\chi_{**},\chi) = U(S;\chi_{**},\chi_*) \cdot U(S;\chi_*,\chi)$$

And dually for software units. The reader can find examples in the illustration. These obvious identities have considerable practical value, permitting us to measure some of the values and derive the others. Usually, certain measures are more difficult to obtain than

others so that with these identities we may avoid the need for taking the more difficult measurements.

Numbers of considerable interest arise when we take sums of utilizations such as:

$$\sum_{*} U(S;\chi_{*},\chi) \text{ where the } \sum_{*}$$

means the sum over the range of the rightmost subscript in $_{*}$. This is the average number of χ_{*}'s executing over the execution time of χ or, in short, an average concurrence. This becomes apparent by simply expanding the definition:

$$M(S;\chi_{*},\chi) = \sum_{*} U(S;\chi_{*},\chi) = \sum_{*} \frac{Tx(S;\chi_{*})}{Tx(S;\chi)} = \frac{\sum_{*} Tx(S;\chi_{*})}{Tx(S;\chi)}$$

i.e., the product of $M(S;\chi_{*},\chi)$ and $Tx(S;\chi)$ yields the sum of the execution times of the χ_{*}'s so that M is the average multiplicity of χ_{*} executions over the time $Tx(S;\chi)$.

$M(S;\chi_{*},\chi)$ is a measure of concurrency of execution of processor subconfigurations which is commonly called the level or degree of multiprocessing. The software dual, $M(S_i,S;\chi)$ is the level of multiprogramming. Consequently, Kolence (KOLE76) refers to them collectively as MP levels. The composite case is the combined multiprogramming-multiprocessing level:

$$M(S_i,S;\chi_{*},\chi) = \frac{\sum_{i,*} Tx(S_i;\chi_{*})}{Tx(S;\chi)}$$

where, of course, the S_i are mutually exclusive, as they are for the simple multiprogramming case.

Often, and often inadvertently or unwittingly, utilizations and MP levels are taken with respect to busy times:

$$Ub(S;\chi_{*},\chi) = \frac{Tb(S;\chi_{*})}{Tx(S;\chi)}.$$

Since $Tb(S;\chi_*) = Tx(S;\chi_*) + Td(S;\chi_*)$,

$$Mb(S;\chi_*,\chi) = \frac{\sum_* Tx(S;\chi_*)}{Tx(S;\chi)} + \frac{\sum_* Td(S;\chi_*)}{Tx(S;\chi)} = M(S;\chi_*,\chi) + Md(S;\chi_*,\chi)$$

which is hardly surprising (the average of sums is the sum of averages). However, under interpretation, this does provide an interesting "cross-over" to results in the operational analysis of queueing networks.

$M(S;\chi_*,\chi)$ is the average number of requests being simultaneously serviced within χ (by the χ_*'s) and $Md(S;\chi_*,\chi)$ the average number of requests waiting for service within χ (e.g., in RPS or seek-delay states). Note that this does not include those requests enqueued before χ. Therefore, $Mb(S;\chi_*,\chi)$, consistently interpreted, is n_x, the number of requests in queue, within χ. (Here, and below, we use the notation of Buzen and Denning for variables from the operational analysis of queueing networks:

n_x	is the number of requests enqued on and being serviced by χ.
X_x	is the throughput rate of χ.
S_x	is the average service time of χ.

In the present context the use of the two notations is unfortunately confusing. Nonetheless, it seems to be the best way to bring out the comparability of the two approaches. If now we introduce C, the number of completions in time $Tx(S;\chi)$ then the average busy time for the χ_*'s is:

$$\overline{Tb}(S;\chi_*) = \frac{\sum_* Tb(S;\chi_*)}{C}.$$

Now, since:

$$Mb(S;\chi_*,\chi) = \frac{\sum_* Tb(S;\chi_*)}{Tx(S;\chi)},$$

it follows that:

$$\bar{T}b(S;\chi_*)\cdot C = Mb(S;\chi_*,\chi)\cdot Tx(S;\chi) \text{ or,}$$

$$\bar{T}b(S;\chi) = Mb(S;\chi_*,\chi)\cdot\frac{Tx(S;\chi)}{C}.$$

The last factor is average service time S_x. This implies that $\bar{T}b(S;\chi)$ is the internal response time (not counting queue-time before χ) R_x, because if $R_x = n_x \cdot S_x$, as the above implies, and if Little's Law, $n_x = R_x \cdot X_x$, where X_x is the throughput rate of χ, then

$$R_x = R_x \cdot X_x \cdot S_x \quad \text{or} \quad X_x = \frac{1}{S_x},$$

a fundamental result (or definition) of queueing network analysis.

The discussion and demonstration given above has significance not just because it shows the isomorphism of the two approaches, under proper interpretation, but because it suggests extensions in the results of queueing network analysis by virtue of the software unit dualities that arise from the software physics approach. This whole matter, the subsuming of queueing network analysis by the methods of software physics, is analyzed and developed by Traister(TRAI79).

MP levels based on execution time, which are a consequence of opportunities for parallelism inherent in the configuration and the patterns of request arrivals, are a measure of how well the software units exploit the configuration or, conversely, how well the configuration meets the requirements of the software units. MP levels based on busy time show the concurrency of waiting as well as executing. In fact, beyond a certain point, as Mb increases, Md increases more rapidly than M, that is, there is more "improvement" in the overlap of waiting than there is in processing. To measure only Mb without obtaining either Md or M is to know only a part of the performance characteristics.

Another set of useful identities arise when we apply the chain rule for utilizations to MP levels:

$$M(S;\chi_{**},\chi) = U(S;\chi_{*},\chi)\cdot M(S;\chi_{**},\chi_{*})$$

and so on.

Ratios of device times to subconfiguration times arise in the analysis of delays due to blocking, especially at control units and channels where connect time is less than drive execution time, as is the case with disk drives. For example, since $U(L;b,\beta)$ is the probability that there is a path block at the control unit when some drive is executing it figures directly in the creation of seek delays and RPS misses.

Ratios of utilizations, such as:

$$\frac{U(S;\chi_{*},\chi)}{U(S;\chi_{*}',\chi)} = \frac{Tx(S;\chi_{*})}{Tx(S;\chi_{*}')}$$

are time balance numbers. They are frequently of interest to configuration designers and system tuners. They are not the only variety of balance indicators, however.

FUNDAMENTAL VARIABLES: WORK

A second fundamental variable is required for the theory to quantify the activity of software units and processors, a measure of how much was effected by them. Within appropriate constraints, this measure must be invariant with respect to time. For instance, if we executed a program against a set of inputs (which determines a software unit) on some configuration and then made a modification which doubled the basic cycle rate of some of the processors and then executed the same program against the same inputs again, the measures of activity must be equal. Similarly, if we execute the same program-input pair on two machines from the same "family", having identical instruction sets, then the measures must be equal even through there may be considerable difference in the internal operations of the two.

This measure is software work. It has the same basis as work in any branch of physics. Work is associated with changes in state. Work is performed whenever there is a change of state: the greater the state change the greater the amount of work and, in discrete state spaces, the minimum amount of work is associated with a minimum state change.

Kolence (KOLE76) develops the definition in the following way. Storages are made up of sets of n-bit containers. The n-bit string in a container at any instant in time is a symbol from an alphabet of 2^n symbols. At every instant in time every container is in a symbol state. A unit of work is performed when the symbol state of one container is changed. A "standard" container size of 8 bits is arbitrarily chosen and, following common usage, both the container and its symbol are called a byte. The working definition then becomes: A unit of work is performed when a processor changes one byte of storage. Since, in practice, our instrumentation will not permit us to see the "before" and "after" values of every processor action on every byte of storage, the "laboratory" definition becomes: One unit of work is performed when a processor transfers one byte to a storage. Clearly, on average, this unit is less than one of the units

in the basic definition because sometimes the byte transferred will be the same as the byte that was in the storage.

We have now defined work and a measure for work by processors. It remains to define them for software units and logical subconfigurations. The work done by an instruction-operand pair is the work done by the processor in executing the instruction on the operands. The amount of work done by a software unit is the sum of the amounts of work done by the instruction-operands that make up the software unit. Work is done by a subconfiguration if work is done at either end of a path going through the subconfiguration. In other words, a β subconfiguration, for example, performs work when there is a write to one of its drives or when there is a read from one or its drives (which changes main storage, which is outside of the β subconfiguration). At one blow, this definition does three things: 1) it defines the work of a subconfiguration as all of the work that "emanates" from it, in either direction, 2) it preserves the containment property for work and 3) it partitions subconfigurations. A partition of a logical subconfiguration is a set of subconfigurations such that their graft is the subconfiguration and such that each "leaf" of the subconfiguration appears in just one member of the set. (A physical leaf having two designations may appear in two members, once under each designation, because the actions of the two logical units are mutually exclusive over time.)

We write the work done by software unit S on subconfiguration χ as $W(S;\chi)$. A unit of work, called simply enough, a work is indicated by ω. Metric prefixes are normally used to indicate larger quantities of work, e.g.:

$$1000\omega = 1 \text{ Kilowork} = 1 \text{ K}\omega; \quad 1000 \text{ K}\omega = 1 \text{ Megawork} = 1 \text{ M}\omega;$$
$$\text{and } 1000 \text{ M}\omega = 1 \text{ Gigawork} = 1 \text{ G}\omega.$$

It follows directly from the definitions given above that

$$W(S;\chi) = \sum_i (S_i;\chi),$$

where the S_i are a partition, and

$$W(S;\chi) = \sum_{*} (S;\chi_{*}),$$

where the χ_{*} are a partition of χ. In particular,

$$W(S;\psi) = W(S;\gamma) + W(S;\emptyset)$$

$$W(S;\emptyset) = \sum_{i} W(S;\chi_i)$$

$$W(S;\chi_i) = \sum_{j} W(S;\beta_{ij}), \text{ and}$$

$$W(S;\beta_{ij}) = \sum_{k} W(S;\delta_{ijk}), \text{ and substituting back}$$

$$W(S;\psi) = W(S;\gamma) + \sum_{i,j,k} W(S;\delta_{yk}).$$

In other words, the total work by S is the simple sum of the CPU work and all the drive work. Another such relationship based on a partition into equipment class subconfigurations often turns out to be very useful:

$$W(S;\psi) = W(S;CPU) + W(S;disk) + W(S;tape) + W(S;ptr) + \ldots$$

This property of software work, the whole is equal to the sum of its parts no matter how the whole is partitioned, is called the extensive property. For the purposes of theory construction it is a very valuable property simply because it permits us to do ordinary arithmetic when dealing with the variable work or functions of it.

WORK AND TIME RELATIONSHIPS: POWER

In every branch of physics there arises naturally a variable which is the rate at which work is done over time. It is called power and is defined

$$P = \frac{dW}{dT},$$

or more simply,

$$P = \frac{W}{T},$$

which is the average power over the interval T. Since work and time are now defined in software physics, the concept of software power also arises naturally.

Stated in its most general terms, software power is defined by the relationship:

$$P(S_i, S; \chi_*, \chi) = \frac{W(S_i; \chi_*)}{Tx(S; \chi)}.$$

In the case where $S_i = S$ then we have the subconfiguration power relationship:

$$P(S; \chi_*, \chi) = \frac{W(S; \chi_*)}{Tx(S; \chi)}.$$

In the case where $\chi_* = \chi$ we have the software unit power relationship

$$P(S_i, S; \chi) = \frac{W(S_i; \chi)}{Tx(S; \chi)}.$$

All of these cases show the work done for some software unit by some subconfiguration relative to some higher clock. They are, therefore, relative powers.

In the case where both the software units and the subconfigurations are equal we have

$$P(S; \chi) = \frac{W(S; \chi)}{Tx(S; \chi)},$$

the work of $(S;\chi)$ with respect to its own clock. This is called absolute power. It is absolute only in the sense that it is not relative. No implication of "ultimate" is intended, for absolute power may be varied, especially in the case of I/O.

A relationship of fundamental importance is revealed by the following sequence of rewritings:

$$P(S;\chi_*,\chi) = \frac{W(S;\chi_*)}{Tx(S;\chi)} \cdot \frac{Tx(S;\chi_*)}{Tx(S;\chi_*)}$$

$$= \frac{W(S;\chi_*)}{Tx(S;\chi_*)} \cdot \frac{Tx(S;\chi_*)}{Tx(S;\chi)}$$

$$= P(S;\chi_*) \cdot U(S;\chi_*\chi)$$

i.e., the relative power equals the absolute power of the lower subconfiguration times the utilization. Dually, for software units, we have

$$P(S_i,S;\chi) = P(S_i;\chi) \cdot U(S_i,S;\chi).$$

When required, the chain rule for utilizations may be employed, such as: $P(S;\chi_{**},\chi) = P(S;\chi_{**}) \cdot U(S;\chi_{**},\chi_*) \cdot U(S;\chi_*,\chi)$ and so on.

Now, observe

$$P(S;\chi) = \frac{W(S;\chi)}{Tx(S;\chi)} = \sum_i \frac{W(S_i;\chi)}{Tx(S;\chi)} = \sum_i P(S_i,S;\chi)$$

where the S_i are a partition of S. Dually we obtain

$$P(S;\chi) = \sum_* P(S;\chi_*,\chi)$$

where χ_*'s are a subconfiguration partition of χ. In other words, the absolute power is the sum of the relative powers of any partition.

The above manipulations suggest that the same sort of thing might be done using ratios of work rather than ratios of time. The ratio

$$\frac{W(S_i,\chi)}{W(S;\chi)}$$

will be written $D(S_i,S;\chi)$ because it is a distribution number, as will become apparent presently. Now, $P(S_i,S;\chi) = D(S_i,S;\chi)\cdot P(S;\chi)$ and dually, $P(S;\chi_*,\chi) = D(S;\chi_*,\chi)\cdot P(S;\chi)$. In other words, the relative power equals the absolute power of the <u>higher</u> subconfiguration times the distribution. We call these distribution numbers because

$$\sum_* D(S;\chi_*,\chi) = \sum_* \frac{W(S;\chi_*)}{W(S;\chi)} = 1,$$

that is the D's are such that $0 \leq D_i \leq 1$ and

$$\sum_i D_i = 1,$$

which are the properties of a frequency distribution.

$$\sum_* P(S;\chi_*,\chi) = \sum_* D(S;\chi_*,\chi)\cdot P(S;\chi) = P(S;\chi)$$

which brings us full circle. Of course, there is also a chain rule for distribution numbers, e.g.,

$$D(S;\chi_{**},\chi) = D(S;\chi_{**},\chi_*)\cdot D(S;\chi_*,\chi).$$

One final set of observations should be made, relating the absolute powers, relative powers, utilizations and distributions:

$$P(S_i,S;\chi_*\chi) = D(S_i,S;\chi_*)\cdot U(S;\chi_*,\chi)\cdot P(S;\chi_*)$$

$$\text{or} \qquad = D(S_i;\chi_*,\chi)\cdot U(S_i,S;\chi)\cdot P(S_i;\chi).$$

The earlier observations, where either $S_i = S$ or $\chi_* = \chi$, can be derived from these last two by simply substituting unity for the appropriate D's and U's. The reader can discover for himself what summing the first equation over i and summing the second over $_*$ produce.

It may be helpful at this point to discuss the significance of these results, in the sense of understanding their impact, relating them to other approaches or intepretations. We have already remarked that utilization in software physics corresponds to utilization in queueing theory. Now let us consider absolute power. $P(S;\chi)$ is the

rate at which χ produces work when it is executing. It is a service rate. Put another way,

$$\frac{1}{P(S;\chi)} = \frac{Tx(S;\chi)}{W(S;\chi)}$$

is the time for χ to complete a unit of work when χ is working. It is the service time of χ. Relative power, $P(S;\chi_*,\chi)$, on the other hand, is the rate at which χ_* produces work over a larger time, an observation time, $Tx(S;\chi)$. It is, therefore, the throughput rate (over the time $Tx(S;\chi)$). Recalling and rewriting one of the first results in this section:

$$U(S;\chi_*,\chi) = P(S;\chi_*,\chi) \cdot \frac{1}{P(S;\chi)}$$

or in the Buzen-Denning notation $U_x = X_x \cdot S_x$, which is the utilization law of operational analysis.

Distribution numbers do not have a simple corresponding operational analysis. They resemble in some ways visit ratios and in other ways routing frequencies. Distribution relative to the work of the full configuration, $W(S;\psi)$ are visit ratios. Distributions with respect to the work of a continuing node, e.g.,

$$\frac{W(S;\chi_*)}{W(S;\chi)}$$

are routing frequencies. Our interest in them will be that they characterize the way the work burden is distributed over subconfigurations. That is, they are a basic device for characterizing workloads, taken separately. Contrariwise, relative powers are a characterization of the way work flow distributes over subconfigurations, taken separately. These observations are the basis for relating the propriety of a configuration for a given workload and vice-versa.

FUNDAMENTAL VARIABLES: STORAGE OCCUPANCY

Storage occupancy is a spatial concept, a measure of the extent of storage associated with a software unit. The unit of measure is a container, in our case, typically, an 8-bit byte. This is both conventional and convenient for our purposes. To establish the association between software units and storage we again appeal to the complete instruction "trace." Associated with each instruction of a software unit is a set of containers and their addresses, typically derived from an initial address for the instruction and a known length in containers for the instruction. Likewise, associated with the operands are sets of containers and their addresses. The amount of storage occupancy is the count of the containers in the set which is the union of the sets of instruction and operands containers. This is the instruction or instantaneous storage occupancy. The storage occupancy of a software unit is the count of containers in the set formed by the union of the sets making up the instruction occupancies. Because the existence of the instructions and operands in the storages constitutes a realization of them, the notation for storage occupancy is $R(S;\sigma)$, where σ is the collection of storages.

Several observations should be made regarding this definition. First, because a software unit consists of instruction executions over some time interval, there is an implicit time interval associated with storage occupancy. The most natural one is from the first instruction execution to the last with respect to the system clock, i.e., $Te(S;\psi)$. Another, of course, is the execution times of the various processors manipulating the storages such as $Tx(S;\psi)$. Now, if the storages are partitioned it is the case that $R(S;\sigma) = \sum_{*} R(S;\sigma_{*})$ where the σ_{*} are a partitioning of σ. Typically, in practice, the partitioning would be into main storages and the storage associated with various drives. Then the total storage occupied by S is the sum of the main storage occupancy and that associated with all the relevant drives.

If the time interval involved is with respect to some higher clock, then we can discuss other software units over the same interval. It should be clear from the previous discussion that over a time two software units may, by turns, occupy some containers, even though the software units are disjoint. To form the composition of storage occupancies we must first form the union of the sets of containers and then take the count. However, this yields the same result as first taking the union of the software units and then measuring the storage occupancy, so that we have

$$R(S;\sigma) = [\overset{U}{\star}\{S_\star;\sigma\}] = [\{\overset{U}{\star}S_\star;\sigma\}] \leq \sum_{\star} R(S_\star;\sigma) \text{ where } S = US_\star$$

which may be a partition or not, and where [] indicates the measure function (count) of the set {}.

The above definition differs from some common measures of storage occupancy. It is usual, thinking in terms of programs instead of software units, to consider the main storage required to contain the loaded program as its main storage occupancy even though some of its instructions are not executed in the consequent run of the program. In our definition, the unexecuted instructions would not appear in any software unit and hence would not contribute to the storage occupancy measure. If one feels compelled to identify the two notions then the instructions of the load process must be considered part of the software unit. Then everything else follows as required. Another common measure is the storage allotted or allocated to a program run. This amount of storage is a function of operating system policy and arbitrary human action. It has no necessary relationship to characteristics arising from the executing system itself and, therefore, cannot be treated in purely software physical terms.

CONCLUDING REMARKS

Only the nuclear theory, the fundamental concepts and primary results, have been presented here. A considerable body of work based on software physics has been done already, chiefly by the staff and members of the Institute for Software Engineering, but it has not been generally published.

In the purely theoretical area, Traister has developed the relationships to operational analysis and queueing theory which are merely suggested in this paper. In the area of applied theory much work has been done on workload forecasting, accounting and related financial issues and capacity analysis and planning, especially of I/O subconfigurations.

In the engineering and "laboratory" areas powers of various CPU's have been measured by both software and hardware monitor techniques, as have a variety of peripheral devices and subsystems. Software physics has proved very valuable in software benchmarking, providing methods of comparing the efficiency of the algorithms themselves, distinct from the influence of hardware as well as the more conventional performance comparisons.

Much remains to be done in extending the theory, in increasing the scope of applications of the theory and in measurement and engineering. There are some very difficult problems in all three areas that have not yet even been approached. It is hoped that this paper will stimulate interest and encourage participation -- for the more hands turned to the tasks, the more likely the successes.

REFERENCES

[DENN78] Denning, P. J. and Buzen, J. P., The Operational Analysis of Queueing Network Models. Computing Surveys, Vol. 10, No. 3 (September 1973) pp. 225-261.

[KOLE70] Kolence, K. W., Measurement and Software Physics. Guide 30, (May 1970)

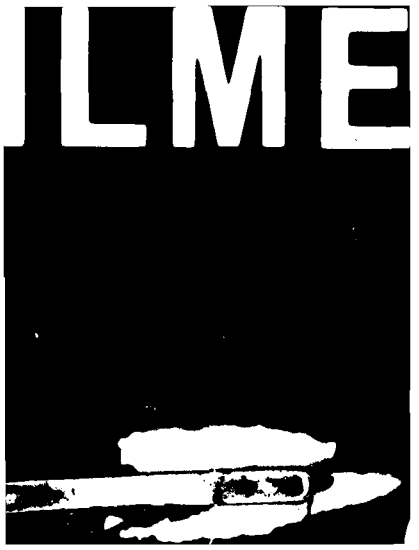
[KOLE72] Kolence, K. W., Software Physics and Computer Performance Measurements. ACM 25th National Conference, (August 1972)

[KOLE73] Kolence, K. W., A Theory for Computer Measurement. Preliminary Report to the National Bureau of Standards, (October 1973)

[KOLE75a] Kolence, K. W., Software Physics. Datamation, (June 1975)

[KOLE75b] Kolence, K. W., The Meaning of Computer Measurements, an Introduction to Software Physics. The Institute for Software Engineering, (1976)

[TRAI79] Traister, Leon M., Queueing Network Analysis in Software Physics. (currently in publication)


LME